JP 2000097451

jc714 U.S. PTO
09/708159
11/08/00

日 本 国 特 許 庁
PATENT OFFICE
JAPANESE GOVERNMENT

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

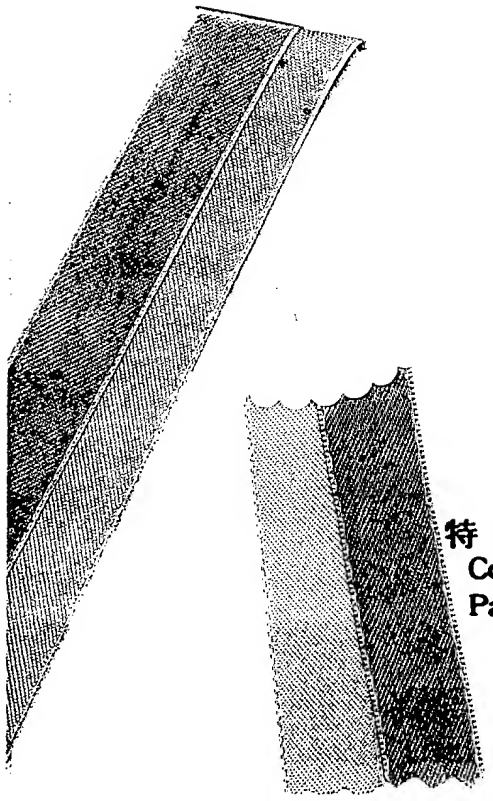
This is to certify that the annexed is a true copy of the following application as filed with this Office.

出 願 年 月 日
Date of Application: １９９９年１１月１７日

出 願 番 号
Application Number: 平成１１年特許願第３２６９９０号

出 願 人
Applicant (s): インターナショナル・ビジネス・マシーンズ・コーポレイション

CERTIFIED COPY OF
PRIORITY DOCUMENT



２０００年 ３月２４日

特許庁長官
Commissioner,
Patent Office 近藤隆彦

出証番号 出証特２０００－３０２１１５１

【書類名】　特許願

【整理番号】　JA999097

【あて先】　特許庁長官　殿

【国際特許分類】　G06F　9/44

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　安江　俊明

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　緒方　一則

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　石崎　一明

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　小松　秀昭

【特許出願人】

【識別番号】　390009531

【住所又は居所】　アメリカ合衆国１０５０４、ニューヨーク州アーモンク（番地なし）

【氏名又は名称】　インターナショナル・ビジネス・マシーンズ・コーポレイション

【代理人】

【識別番号】　100086243

【弁理士】

【氏名又は名称】　坂口　博

【連絡先】　０４６２－７３－３３１８、３３２５、３４３１

【選任した代理人】

【識別番号】　100091568

【弁理士】

【氏名又は名称】　市位　嘉宏

【手数料の表示】

【予納台帳番号】　024154

【納付金額】　21,000円

【提出物件の目録】

【物件名】　明細書　1

【物件名】　図面　1

【物件名】　要約書　1

【包括委任状番号】　9304391

【包括委任状番号】　9304392

【プルーフの要否】　要

【書類名】　明細書

【発明の名称】　プログラム実行方法

【特許請求の範囲】

【請求項１】

実行途中のメソッドから、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行うプログラム実行方法であって、

遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、

遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、

前記コードの移動、プライベータイゼーションおよびコモンサブエキスプレッションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、

遷移処理において再計算を実施するステップと

を含むプログラム実行方法。

【請求項２】

前記メソッドにおいて、インタプリタ処理とコンパイル処理との間で遷移した方が、遷移しない場合に比べて実行が高速化する点を新たな遷移点にするステップをさらに含む

請求項１に記載のプログラム実行方法。

【請求項３】

前記遷移点それぞれにおいて、インタプリタ処理とコンパイル処理との間で遷移するために必要な情報を生成するステップと、

前記生成した情報を前記遷移点それぞれに対応づけて記憶するステップと

をさらに含み、

前記再計算を実施するステップにおいて、前記遷移点それぞれにおいて、前記対応づけられて記憶された情報を用いて再計算を行う

請求項１または２に記載のプログラム実行方法。

【請求項４】

実行途中のメソッドを、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行って実行するプログラムであって、

遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、

遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、

前記コードの移動、プライベータイゼーションおよびコモンサブエキスプレッションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、

遷移処理において再計算を実施するステップと

を含むプログラムを媒介する媒体。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】

本発明は、Ｊａｖａ等の言語で作成されたプログラムを動的にコンパイルするプログラム実行方法に関する。

【０００２】

【従来の技術】

近年のＪａｖａの普及により動的コンパイラを備える環境の重要性が増している。動的コンパイラによる実行環境では，初期実行時に全てのメソッドを時間のかかる最適化コンパイルしてしまうと、コンパイル時間により逆に速度低下を引き起こすプログラムも存在する。このため最初の実行はインタプリタ実行や最適化を行わない高速コンパイルされた非最適化コードを利用した実行を行い、メソッド単位で起動回数情報やメソッドに多数回繰り返されるループが存在するなどの情報を収集し、それらの情報をもとに最適化コンパイルを実施しすることで全体として高速実行を実現している。

【０００３】

例えば、従来から「Adaptive Compilation [HCU91]」と呼ばれる方法が知られている。

この方法は、メソッドの最初の実行時にはほとんど最適化しないでコンパイルを行い、各メソッド毎にinvocation counterを使って起動回数をカウントし、値が閾値を超えたメソッドに対して最適化レベルを上げて再コンパイルを実施する方法である。

本手法では再コンパイルされたメソッドが実行されるのはそのメソッドを呼び出すタイミングのみのため、１で述べた１回しか起動されないが非常に多数回繰り返される時間のかかるループ（最適化対象とするループであり、多重ループを含む）が存在するメソッドに対して効果がない問題がある。

[HCU91] Urs Hoelzle, Craig Chambers, and David Ungar. Optimizaing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In ECOOP'91 Conference Proceedings, pp.21-38, Geneva, Switzerland, July 1991. Published as Springer Verlag LNCS 512, 1991.

【０００４】

また、従来から「Deoptimization [HCU92]」と呼ばれる方法が知られている。この方法は、最適化したメソッドのコードをデバッグするために最適化されていないコードを作成し、そのコードに実行を遷移させてデバッグするための方法である。この方法では、（１）最適化コンパイルする際にメソッドの先頭とループの後方分岐命令の２種類の位置にinterrupt pointを設定し、（２）interrupt pointを超えた最適化を行わない制約を加えた最適化コンパイルを行う。（３）実行中にinterruptがかかった場合はinterrupt point単位でプログラムをresumeし、（４）interrupt pointのスタックフレームを保存する。（５）メソッドを最適化せずにコンパイルし、（６）スタックフレームを最適化されない状態に合わせて再現してinturrupt状態に入る。本手法では、実行途中での遷移という点で本発明と類似するが、遷移対象が非最適化コードであり遷移点でのレジスタイメージなどの実行状態がソースコードから一意に決定できるため、本手法は遷移先の状態が複雑化する最適化コードへの遷移にはそのままでは適用できない問題がある。

[HCU92] Urs Hoelzle, Craig Chambers, and David Ungar. Debugging Optimized Code With Dynamic Deoptimization. In Proceedings of the SIGPLAN '92

Conference on Programming Language Design and Implelementation, pp.21-38, June 1992.

【０００５】

また、従来から「Dynamic recompilation [HU94]」と呼ばれる方法が知られている。

この方法は、最適化されていないメソッドの実行途中から最適化コンパイルされたメソッドに、メソッドの実行途中で遷移する手法の概要を示している。この方法では、（１）再コンパイルするメソッドを検出し、（２）実行を再開する点に印を設定し、（３）その点での全ての有効なレジスタが指している値を計算する。これらが成功した場合は（４）スタック上の最適化されていないメソッドを最適化されたメソッドに置き換える。置き換えできない場合は、生成されたコードは後続するそのメソッドの呼び出しで使用されることとなる。本手法はその詳細な処理が記述されていないが、上述した「Deoptimization [HCU92]」の逆処理であると述べられていることから、問題点は以下の通りとなる。

【０００６】

まず本手法では複数の遷移点を扱っていない。また遷移点を超えた最適化を行っていない点、メモリの増加を考慮していない点である。

[HU94] Urs Hoelzle and David Ungar. Optimizing Dynamicall-Dispatched Calls with Run-Time Type Feedback. In Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, pp.326-336. Published as SIGPLAN Notices 29(6), June 1994.

【０００７】

最適化コンパイルを実施した際の実行の切り替え方法は、次のメソッド呼び出しで切りかえる方法と、メソッドの実行途中から最適化コードの実行に遷移する方法の２方法がある。特に後者の方法は、１回しか起動されないが非常に多数回繰り返される時間のかかるループが存在するメソッドの処理を高速化するためには必須の機能である。しかしながらこの実行途中での切り替えに対する従来手法では、以下の２つのような問題が存在した。

【０００８】

［問題１］切り替えを行うプログラム上の位置（以下遷移点と呼ぶ）を超えて最適化を実施していない。このため遷移点が存在すると、遷移点が存在しない場合に生成されたコードと比べてコードの質が低下する。後続する当該メソッドの呼び出しによる実行は、遷移点がない状態で最適化コンパイルされた場合に比べて実行速度が低下してしまう問題が生じる。

特に遷移点がループ中に存在する場合は、実行性能に大きく貢献するループ最適化の実施ができなくなるため性能低下が著しくなる。

【０００９】

［問題２］マルチ･スレッド環境において複数の異なる遷移点から遷移する場合を想定しておらず、遷移が発生する度にコンパイルを実施している。このため複数回のコンパイル時間が実行速度の低下を引き起こすとともに、それらのコードが重複して存在してメモリ使用量が増大する問題が生じる。

【００１０】

【発明が解決しようとする課題】

本発明は、上述した従来技術の問題点に鑑みてなされたものであり、切り替えを行うプログラム上に遷移点がある場合であっても、遷移点を超えたより高度な最適化が可能なプログラム実行方法を提供することを目的とする。

本発明は、マルチスレッド環境において複数の異なる遷移点から遷移する場合であっても、複数回のコンパイルが不要で、しかも、発生したコードに重複を生じさせないプログラム実行方法を提供することを目的とする。

【００１１】

【課題を達成するための手段】

上記目的を達成するために、本発明に係るプログラム実行方法は、実行途中のメソッドから、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行うプログラム実行方法であって、遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、前記コードの移動、プライベータイゼーションおよびコ

モンサブエキスプレっションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、遷移処理において再計算を実施するステップとを含む。

【００１２】

好適には、前記メソッドにおいて、インタプリタ処理とコンパイル処理との間で遷移した方が、遷移しない場合に比べて実行が高速化する点を新たな遷移点にするステップをさらに含む。

【００１３】

好適には、前記遷移点それぞれにおいて、インタプリタ処理とコンパイル処理との間で遷移するために必要な情報を生成するステップと、前記生成した情報を前記遷移点それぞれに対応づけて記憶するステップとをさらに含み、前記再計算を実施するステップにおいて、前記遷移点それぞれにおいて、前記対応づけられて記憶された情報を用いて再計算を行う。

【００１４】

また、本発明に係る媒体は、実行途中のメソッドを、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行って実行するプログラムであって、遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、前記コードの移動、プライベータイゼーションおよびコモンサブエキスプレっションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、遷移処理において再計算を実施するステップとを含むプログラムを媒介する。

【００１５】

【発明の実施の形態】

［本発明の要点］

まず、本発明の要点を説明する。

【００１６】

［遷移点の存在によってコードの質が低下させないための方法］

遷移点でコードの質が低下する原因は、２つ存在する。１つはループ中に遷移点が存在すると多くのループ最適化処理を施せなくなる点である。またもう１つは、コードの移動およびprivatizationやcommon subexpression eliminationのように、計算途中の状態をレジスタやメモリのローカル領域に保持しておく最適化手法が実施できなくなる点である。これらの問題に対して次のような方法を用いる。

【００１７】

（１）遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合は、コンパイルコードにおける遷移点をループの入り口に移動する。

（２）遷移点がループ内部に位置する場合、遷移点の存在がループ最適化に影響を与えないようにするために、ループの入り口から、ループの入り口と遷移点をともにpost-dominateする点までをループ入り口の直前に複製し、遷移点がループの外側もしくはループの入り口と等しくなるようにループを変形する。

（３）コードの移動およびprivatizationやcommon subexpression eliminationが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせておき、遷移処理において再計算を実施する。

【００１８】

［１つのコードで通常の呼び出しと途中からの遷移に対応する方法］

インタプリタからこれから遷移しようとする点とともに、メソッドのプログラムを検索して遷移する可能性がありかつ遷移により高速化の効果のある点を検出し、これらの点も遷移点とする。

各遷移点に対して、その遷移点でのレジスタ使用状態、再計算コード情報からなる遷移情報を生成し、元のコード上のアドレスとあわせて、これらを要素とするテーブルを生成し、メソッドからアクセスできる場所に設定する。遷移実行時には、遷移点のアドレスをキーとしてこのテーブルをアクセスし、遷移情報を取得する。

【００１９】

インタプリタが使用しているスタックフレームからコンパイルされたコードで使用するスタックフレームへの切替時に不要な変数領域を生成しない方法。インタプリタでは各メソッドで指定されたローカル変数の数全てがスタックフレーム中に割り当てられる。一方コンパイルされたコードでは、コンパイルされたコードでは、複数のローカル変数が同一の退避場所を使用していたり、レジスタのみに値が保持されスタックフレーム上へは退避されない変数が存在したりする。このためメソッドで指定したローカル変数全てがスタックフレーム内に割り当てられるわけではない。この問題に対して次の方法を用いる。

【００２０】

コンパイルされたコードが必要とするスタックフレーム中のローカル変数領域の大きさを、コンパイル時にメソッドからアクセスできる場所に格納しておく。このとき領域の大きさは、コンパイルされたコードが必要とする分のみを割り当てる。また、遷移情報には、遷移点で生きている変数に対してそれがレジスタに保持されているか、スタックフレーム中に保持されているかの情報を持たせる。遷移時には、まず、インタプリタが使用しているスタックフレームを退避し、コンパイルされたコードのためのスタックフレームを生成した後、遷移情報を使用して必要なローカル変数の値をレジスタ上かスタックフレーム中にコピーする。最後にインタプリタのスタックフレームを除去して、遷移点に実行を移す。

インタプリタから機械コード実行への切り替えが、動的コンパイラを起動した最初の１回の場合に限るときは、緩衝コードをスタック領域に生成することで緩衝コードに対するメモリ使用の増加をなくすことができる。

【００２１】

以下、本発明の手順を説明する。

［３．１．インタプリタが途中遷移を決定した場合］

インタプリタが途中遷移を決定した場合、以下の処理を行う。

（３．１．１）既にメソッドがコンパイルされている場合は、メソッドより遷移情報テーブルを取得し、遷移点の命令のアドレスから遷移情報の取得を試みる。遷移情報を取得できた場合は、３．３の２以降の処理を行う。遷移情報を取得できなかった場合は、この点での遷移処理を中止しインタプリタによる実行を継続

する。

【００２２】

（３．２．１）動的コンパイラを起動する。このとき遷移を決定した命令のアドレスと遷移後に実行される命令のアドレスをコンパイラに渡す。遷移後に実行される命令のアドレスが自明な場合にはこれを省略することもできる。

【００２３】

［３．２動的コンパイラの処理］

動的コンパイラで以下の処理を行う。

（３．２．１）メソッドのコードを、Basic Block（以下ＢＢと省略する）に分割し、Control Flow Graphを生成する。このとき、インタプリタが指定した遷移点がBasic Blockの先頭となるようにする。

【００２４】

（３．２．２）インタプリタが指定した遷移点を含むＢＢに印を付ける。

【００２５】

（３．２．３）メソッドのコードで、インタプリタが指定した遷移点以外に遷移される可能性のあり、かつ遷移による効果がある遷移候補点を探索する。この処理はインタプリタが指定した遷移点以外を遷移点としない場合は実施しない。遷移候補点はメソッド中のインタプリタが検出しうる全位置から遷移効果がないとコンパイラが判断可能な位置を除いたものとする。遷移効果がない点とは、ループ外の点などである。

【００２６】

（３．２．４）遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合は、コンパイルコードにおける遷移点をループの入り口に移動する。移動上の問題がある場合とは、ヒープ領域への書きこみ命令が存在する場合、およびループの入り口から遷移点の間にある変数が定義されており、ループの入り口からその定義点までの間にその変数の参照が存在し、かつループの入り口においてその変数の値を再計算することができない場合である。再計算可能な場合は、遷移点に対する再計算情報を生成する。

【００２７】

（３．２．５）遷移点がループ内部に位置する場合。

遷移点がループ内部に位置する場合、遷移点の存在がループ最適化に影響を与えないようにするために、遷移点がループ（最適化対象とするループであり、多重ループを含む）の開始点となるようにループを変形する。変形方法は、ループのentry pointから、ループのentry pointと遷移点をともにpost-dominateするループ内の点までのコードを複製する操作を行う。ここで複製されるコードは最悪の場合、そのループのコードよりも小さい。さらにインタプリタから遷移される点がループの後方分岐命令に限定される場合は、Ｊａｖａのforループではループの制御コードのみを複製すればよく、do-whileループではまったく複製する必要がない。

【００２８】

（３．２．６）コードの移動やcommon subexpressionとprivatizationによる値のキャッシングが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報をそれらが超えた全ての遷移点に登録する。

【００２９】

（３．２．７）メソッドに対するコードを生成する。

【００３０】

（３．２．８）コンパイルされたコードで使用するローカル変数領域の大きさをメソッドからアクセスできる場所に格納する。

【００３１】

（３．２．９）各遷移点に対して、遷移点のコードアドレス、キャッシュの再計算コード情報、および遷移点のレジスタ使用情報からなる遷移情報を生成し、元のコード上の遷移命令のアドレスとついにした遷移情報テーブルを作成する。このテーブルをメソッドからアクセスできる場所に格納する。遷移情報は、実際のコードとしてあらわしてもよいし、共通の遷移ルーチンに渡すデータとして表してもよい。またインタプリタから機械コード実行への切り替えが動的コンパイラを起動した最初の１回の場合に限るときは、遷移情報をスタック領域に生成することで、遷移が終わったときに即時除去してしまうことも可能である。

【００３２】

［３．３．インタプリタの処理］

インタプリタで以下の処理を行う。

（３．３．１）動的コンパイルの結果よりコンパイルが成功した場合は、遷移情報テーブルより遷移情報を取得する。コンパイルが失敗した場合は、この点での遷移処理を中止しインタプリタによる実行を継続する。

〔００３３〕

（３．３．２）コンパイルされたコードで使用するローカル変数領域の大きさを獲得し、インタプリタで使用していたスタックフレーム中の情報のうち必要なものを退避してからコンパイルされたコード用のスタックフレームを生成する。

〔００３４〕

（３．３．３）遷移情報を使用してコンパイルされたコード用のスタックフレームに必要な情報を設定する。遷移情報がコードで表される場合はコードを実行する。遷移情報がデータとして表される場合は、遷移処理ルーチンにこのデータを渡して遷移処理を実施する。遷移処理の手順としては、（１）スタック領域に格納される値を退避領域からコピーする、（２）再計算情報により値の再計算を行い、結果をローカル変数かレジスタに設定するか退避領域に設定する、（３）レジスタに値をロードする、などにより実現できる。

〔００３５〕

（３．３．４）遷移点のアドレスより実行を開始する。

〔００３６〕

［プログラム実行装置１］

図１は、本発明に係るプログラム実行装置１の構成を示す図である。

図１に示すように、プログラム実行装置１は、サーバ１０とクライアント１２とが、ネットワーク１０６を介して接続されて構成される。

サーバ１０は、Ｊａｖａソースコード１００、Ｊａｖａバイトコードコンパイラ（ＪＡＶＡＣ）１０２およびＪａｖａバイトコード１０４を含む。

〔００３７〕

クライアント１２は、記録媒体１４０を介してクライアント１２に供給され、記憶装置１４を介してハードウェア１３２で実行されるＪａｖａ処理プログラム

１２０およびＯＳ１３０を含む。

Ｊａｖａ処理プログラム１２０は、Ｊａｖａバイトコードベリファイヤ１２２、Ｊａｖａインタプリタ１２４、ＪＩＴコンパイラ１２６およびネイティブコードエキスキューション１２８を含む。

【００３８】

サーバ１０において、Ｊａｖａソースコード１００は、ＪＡＶＡＣ１０２などのバイトコードコンパイラによりバイトコード１０４に変換され、ネットワーク１０６を介してクライアント１２に供給される。

【００３９】

クライアント１２において、バイトコードベリファイヤ１２２は、サーバ１０からのバイトコード１０４を検証する。

【００４０】

Ｊａｖａインタプリタ１２４は、ＪＩＴコンパイラ１２６が、検証されたバイトコード１０４をコンパイルしない場合に、検証されたバイトコード１０４を実行する。

【００４１】

ＪＩＴコンパイラは、検証されたバイトコード１０４をコンパイルする場合には、バイトコード１０４をコンパイル処理してネィティブコードを生成する。

【００４２】

ネイティブコードエキスキューション１２８は、コンパイル処理の結果として生成されたネィティブコードを実行する。

【００４３】

ここでは、Ｊａｖａインタプリタ１２４とＪＩＴコンパイラ１２６への組み込む場合に対する実施例を示す。

実施例で使用する条件は以下の通りである。

Ｊａｖａインタプリタ１２４はループ（最適化対象とするループであり、多重ループを含む）の後方分岐となる命令でのみ遷移を行う。遷移を行わないことに決めた命令にはフラグを設定し、後続するThreadで遷移しないようにする。

遷移後は、遷移を決定した後方分岐命令の分岐先命令から実行を開始するもの

とする。

〔００４４〕

Ｊａｖａインタプリタ１２４では遷移点を、遷移先のアドレスではなく遷移を決定した後方分岐命令で行う。

ＪＩＴコンパイラ１２６では、ＪＩＴコンパイラ１２６を起動したＪａｖａインタプリタ１２４が遷移を決定した後方分岐命令に加えて、他の後方分岐命令のうちでフラグが設定されておらずかつ最適化対象となった

ループの後方分岐命令を遷移点として検出する。

遷移点をループ外へ移動するために複製されるコードの量を少なくするために、遷移点をより外のループは最適化対象としない。この制限は、次の点で妥当である。まず外側のループがforループの場合、Ｊａｖａインタプリタ１２４によりループの条件判定を１度以上実施しているので、その段階で遷移しないことから繰り返し回数が少ないループと考えられる。

遷移情報は実行コードとして生成する。

ＪＩＴコンパイラ１２６で生成されるコードでは、元のプログラムに含まれるローカル変数領域は必ずスタックフレーム上に割り当てられるものとする（説明を簡単にするため）。

〔００４５〕

なお、実施例中では以下の用語を使用する。

transfer point ... Ｊａｖａインタプリタ１２４からＪＩＴコンパイラ１２６に遷移する点

transfer bwd jump ... Ｊａｖａインタプリタ１２４でtransfer pointを検出した後方分岐命令

transfer bwd jump address ... 実行中のメソッドのbytecode中のtransfer bwd jump命令が格納されているメモリのアドレス

transfer point table ... transfer bwd jump addressと遷移のためのpad codeのentry addressの組を要素とするテーブル

メソッド情報（以下mbと略す） ... Ｊａｖａでメソッドの情報をもつ構造体。

メソッドコンパイル情報（以下minfoと略す） ... コンパイル時にメソッドに関するさまざまな情報を蓄積する構造体。

〔００４６〕

［Ｊａｖａインタプリタ１２４の処理（処理１）］

Ｊａｖａインタプリタ１２４－で、ＪＩＴコンパイラ１２６に即時遷移するbackedgeを検出する。

必要があればレジスタにキャッシュしている値をメモリに書き出す。

ＪＩＴコンパイラ１２６を呼んでコンパイルを実行する。このとき、遷移のきっかけとなったbackedgeのアドレスをＪＩＴコンパイラ１２６に渡す。

〔００４７〕

［ＪＩＴコンパイラ１２６の処理（処理２）］

コンパイルを排他処理するためのロックをかける。

transfer bwd jump addressが渡された場合で既にコンパイル済の場合、遷移点テーブルを検索し、該当する遷移点アドレスがテーブル上に存在する場合、そのテーブル・オフセットを戻り値として返す（終了）。

transfer bwd jump addressが渡された場合、メソッド情報に遷移点に対する処理を実施することを表すフラグ（以下このフラグをgen_tpフラグと呼ぶ）を設定する。

ループの後方分岐の数をカウントし、transfer point tableをアロケートする。

Bytecodeをtraverseし、Basic Block（以下ＢＢと呼ぶ）およびループを構築する。この際、３のフラグが設定されている場合には、以下の処理を行う。

〔００４８〕

Ｊａｖａインタプリタ１２４から受け取ったtransfer bwd jumpから得たtransfer pointのＢＢをメソッド情報に登録する。

ＢＢに遷移点をあらわすフラグを立てる。このフラグが立っているＢＢは前者ＢＢなどとマージしては行けない。

ループを形成する後方分岐で、遷移しないことを示すフラグがたっていない命令をtransfer point tableに登録する。

【００４９】

ループにTransfer pointが含まれる場合は、以下の処理を行う。

Control flow graphに従って、ループの入り口から遷移点までのコードをトレースする。トレース時には、次の命令の存在をチェックする。これらの命令が存在しない場合はTransfer pointをループ入り口まで移動する。

【００５０】

つまり、Control flow graphにしたがって、ループの入り口から遷移点までのコードをトレースする。

次に、トレース時に、次の命令の存在をチェックする。これらの命令が存在しない場合は、Transfer pointをループ入り口に変更し、ループ外への移動処理を終了するヒープ領域への書き込み命令を実行する。

次に、ローカル変数の定義命令を実行する。

さらに、ループ入り口より遷移店に至るコードをループ外に複製し、遷移点をループ外へ移動する。

【００５１】

common subexpression eliminationを実施する。

Transfer pointをまたがる場合、除去される計算がＪａｖａインタプリタ１２４がもつローカル変数から再計算できる場合は、再計算情報としてそのTransfer pointに登録する。再計算できない場合は、そのTransfer pointをこえて処理を実施しないように制御する。

【００５２】

field変数に対するprivatizationを実施する。

Transfer pointをまたがる場合、除去される計算がＪａｖａインタプリタ１２４がもつローカル変数から再計算できる場合は、再計算情報としてそのTransfer pointに登録する。再計算できない場合は、そのTransfer pointをこえて処理を実施しないように制御する。

【００５３】

コード生成では以下の処理を行う。

minfoのtransfer point tableの情報を使って、mbからアクセスできる場所に

、transfer pointテーブル{(元のコードのアドレス，遷移コード)を要素とする配列}と、テーブルサイズ格納領域を作成し、transfer pointのbytecode addressを設定する。

〔００５４〕

このとき、検索を容易にするためbytecode addressを小さい順に並べ替えて生成する。

transfer point tableの情報とtransfer pointのレジスタイメージを用いて、キャッシュの再計算コードと各transfer pointのための遷移コード（メモリ上の値をレジスタに載せるコード）を生成し、transfer pointテーブルに transfer pointコードアドレスとして設定する。

ＪＩＴコンパイラ１２６が必要とする作業領域サイズ（ローカル変数領域とスタック領域）をmbからアクセスできる場所に格納する。

〔００５５〕

［Ｊａｖａインタプリタ１２４側の処理（処理３）］

コンパイラの戻り値をチェックする。コンパイルに失敗していたら、Ｊａｖａインタプリタ１２４実行を継続する。（終了）

mbのtransfer point tableを取得し、現在のbwd jump addressがtable上にあるかどうかを調べる。table上にaddressがなければ、Ｊａｖａインタプリタ１２４実行を継続する。（終了）

ＪＩＴコンパイラ１２６から受け取ったワークエリアサイズを加味して、スタックフレームをＪａｖａインタプリタ１２４用のものからＪＩＴコンパイラ１２６用のものに変換する。

〔００５６〕

mbからtransfer pointテーブルを獲得する。

transfer pointテーブルと、現在実行しているbytecode addressからtransfer pointコードアドレスを獲得する。

transfer pointコードアドレスより実行を開始する。

〔００５７〕

以下、さらに図２～図８に示すフローチャートを参照して、本発明に係るプロ

グラム実行装置１の処理を説明する。

本発明は、バイトコード実行処理（図２；Ｓ１０）において、Ｓ１４の処理として位置づけられる。

図２に示すように、ステップ１００（Ｓ１００）において、プログラム実行装置１は、メソッドを実行する。

ステップ１０２（Ｓ１０２）において、プログラム実行装置１は、メソッドをインタプリタで実行するか否かを判断し、インタプリタで実行する場合にはＳ１０４の処理に進み、これ以外の場合にはＳ１１４の処理に進む。

【００５８】

ステップ１０４（Ｓ１０４）において、プログラム実行装置１は、インタプリタでの処理を開始する。

【００５９】

ステップ１０６（Ｓ１０６）において、プログラム実行装置１は、命令を読み込む。

【００６０】

ステップ１０８（Ｓ１０８）において、プログラム実行装置１は、コンパイルされたコードを実行するか否かを判断し、実行する場合にはＳ１４のＳ１６の処理に進み、これ以外の場合にはＳ１１０の処理に進む。

【００６１】

ステップ１１０（Ｓ１１０）において、プログラム実行装置１は、コードをインタプリタで実行する。

【００６２】

ステップ１１２（Ｓ１１２）において、プログラム実行装置１は、インタプリタでの処理を終了する。

【００６３】

ステップ１１４（Ｓ１１４）において、プログラム実行装置１は、コンパイラによりメソッドをコンパイルする。

【００６４】

ステップ１１６（Ｓ１１６）において、プログラム実行装置１は、コンパイル

により生成されたネィティブを実行する。

【００６５】

図３は、図２に示した本発明に係るＳ１４の処理を示す図である。

図３に示すように、Ｓ１４内の遷移のための前処理（Ｓ１６）のステップ１６０（Ｓ１６０）において、プログラム実行装置１は、メソッドがコンパイル済みか否かを判断し、コンパイル済みの場合にはＳ１６２の処理に進み、これ以外の場合にはＳ２０（図４～図８を参照して後述）の処理に進む。

【００６６】

ステップ１６２（Ｓ１６２）において、プログラム実行装置１は、メソッドより遷移情報テーブルを取得する。

【００６７】

ステップ１６４（Ｓ１６４）において、プログラム実行装置１は、遷移点の命令アドレスをキーとして遷移情報テーブルを検索する。

【００６８】

ステップ１４２（Ｓ１４２）において、プログラム実行装置１は、遷移情報が存在するか否かを判断し、存在する場合には、遷移処理（Ｓ１８）に進み、これ以外の場合にはＳ１４６の処理に進む。

【００６９】

ステップ１４４（Ｓ１４４）において、プログラム実行装置１は、Ｓ２０の処理においてコンパイルが成功したか否かを判断し、成功した場合にはＳ１８の処理に進み、これ以外の場合にはＳ１４６の処理に進む。

【００７０】

ステップ１４６（Ｓ１４６）において、プログラム実行装置１は、インタプリタでの実行処理に戻る。

【００７１】

ステップ１４８（Ｓ１４８）において、プログラム実行装置１は処理を終了する。

【００７２】

遷移処理（Ｓ１８）のステップ１８０（Ｓ１８０）において、プログラム実行

装置１は、遷移情報テーブルから遷移情報を取得する。

【００７３】

ステップ１８２（Ｓ１８２）において、プログラム実行装置１は、現在のスタックフレーム内の必要情報を退避する。

【００７４】

ステップ１８４（Ｓ１８４）において、プログラム実行装置１は、遷移情報を用いてスタックフレームのサイズを変更する。

【００７５】

ステップ１８６（Ｓ１８６）において、プログラム実行装置１は、遷移情報と退避情報からネィティブコード用のスタックフレームを生成する。

【００７６】

ステップ１８８（Ｓ１８８）において、プログラム実行装置１は、遷移情報より遷移アドレスを取得し、そこより実行を開始する。実行完了後、プログラム実行装置１は、処理を終了する（Ｓ１４８）。

【００７７】

図４は、図３に示した遷移のためのコンパイル処理（Ｓ２０）を示すフローチャートである。

Ｓ２０のステップ２０２（Ｓ２００）において、プログラム実行装置１は、メソッドのコードをベーシックブロック（Basic Block）に分割する。

【００７８】

ステップ２０４（Ｓ２０４）において、プログラム実行装置１は、遷移点がベーシックブロックの先頭か否かを判断し、先頭の場合にはＳ２０８の処理に進み、これ以外の場合にはＳ２０６の処理に進む。

【００７９】

ステップ２０６（Ｓ２０６）において、プログラム実行装置１は、遷移点がベーシックブロックの先頭になるようにベーシックブロックを分割する。

【００８０】

ステップ２０８（Ｓ２０８）において、プログラム実行装置１は、制御フローグラフの生成を行う。

ステップ２１０（Ｓ２１０）において、プログラム実行装置１は、遷移点のベーシックブロックに印を設定し、他の遷移可能点の検索処理（Ｓ２２；図５）の処理に進む。

〔００８１〕

図５は、図４に示した他の遷移可能点の検索処理（Ｓ２２）を示すフローチャートである。

図５に示すように、Ｓ２２のステップ２２２（Ｓ２２２）において、プログラム実行装置１は、メソッド内の遷移点以外のコードに対して繰り返す処理を開始する。

〔００８２〕

ステップ２２４（Ｓ２２４）において、プログラム実行装置１は、インタプリタから遷移する可能性がある命令であるか否かを判断し、可能性がある場合にはＳ２２６の処理に進み、これ以外の場合にはＳ２３６の処理に進む。

〔００８３〕

ステップ２２６（Ｓ２２６）において、プログラム実行装置１は、遷移する効果があるかどうかを解析する。

〔００８４〕

ステップ２２８（Ｓ２２８）において、プログラム実行装置１は、遷移効果があるか否かを判断し、ある場合にはＳ２３０の処理に進み、これ以外の場合にはＳ２３２の処理に進む。

〔００８５〕

ステップ２３０（Ｓ２３０）において、プログラム実行装置１は、遷移点がベーシックブロックの先頭か否かを判断し、先頭である場合にはＳ２３４の処理に進み、これ以外の場合にはＳ２３２の処理に進む。

〔００８６〕

ステップ２３２（Ｓ２３２）において、プログラム実行装置１は、遷移点がベーシックブロックの先頭になるようにベーシックブロックを分割し、制御フローグラフを修正する。

〔００８７〕

ステップ２３４（Ｓ２３４）において、プログラム実行装置１は、遷移点のベーシックブロックに印を設定する。

【００８８】

ステップ２３６（Ｓ２３６）において、プログラム実行装置１は、メソッドの中の遷移点以外のコードに対して繰り返す処理を終了する。

【００８９】

ステップ２３８（Ｓ２３８）において、プログラム実行装置１は、Ｓ２２の処理を終了して、ループを変形せずにループ内の遷移点のループ外への移動処理（Ｓ２６；図６）を行う。

【００９０】

図６は、図４に示したループを変形せずにループ内の遷移点のループ外への移動処理（Ｓ２６）を示すフローチャートである。

図６に示すように、Ｓ２６のステップ２６２（Ｓ２６２）において、プログラム実行装置１は、Ｓ３００までのループ１の処理を、各遷移点ごとに繰り返す。

【００９１】

ステップ２６４（Ｓ２６４）において、プログラム実行装置１は、遷移点が最適化可能なループの中に位置するか否かを判断し、位置する場合にはＳ２６６の処理に進み、これ以外の場合にはＳ３００の処理に進む。

【００９２】

ステップ２６６（Ｓ２６６）において、プログラム実行装置１は、ループ内にメソッド以外からも参照可能な領域への書き込み命令が存在するか否かを判断し、存在する場合にはＳ３００の処理に進み、これ以外の場合にはＳ２６８の処理に進む。

【００９３】

ステップ２６８（Ｓ２６８）において、プログラム実行装置１は、ループ外で定義された値がループ内で参照されるローカル変数の集合Ｖｒを計算する。

【００９４】

ステップ２７０（Ｓ２７０）において、プログラム実行装置１は、ループ内で定義されるローカル変数の集合Ｖｄを計算する。

【００９５】

ステップ２７２（Ｓ２７２）において、プログラム実行装置１は、Ｓ２８２までのループ２の処理を、再計算が必要なローカル変数の集合（Ｖｒ∩Ｖｄ）の各変数ＶＩに対して繰り返す。

【００９６】

ステップ２７４（Ｓ２７４）において、プログラム実行装置１は、ループ開始点でＶＩを再計算するための情報を収集する。

【００９７】

ステップ２７６（Ｓ２７６）において、プログラム実行装置１は、ループ開始点においてＶＩを再計算することが可能か否かを判断し、可能な場合にはＳ２７８の処理に進み、これ以外の場合にはＳ３００の処理に進む。

【００９８】

ステップ２７８（Ｓ２７８）において、プログラム実行装置１は、ＶＩの再計算に必要なローカル変数の集合Ｖｃを計算する。

【００９９】

ステップ２８０（Ｓ２８０）において、プログラム実行装置１は、集合（Ｖｃ∩Ｖｄ）が空集合であるか否かを判断し、空集合である場合にはＳ２８２の処理に進み、これ以外の場合にはＳ３００の処理に進む。

【０１００】

ステップ２８４（Ｓ２８４）において、プログラム実行装置１は、遷移点をループの開始点に移動する。

【０１０１】

ステップ２８６（Ｓ２８６）において、プログラム実行装置１は、集合（Ｖｒ∩Ｖｄ）が空集合であるか否かを判断し、空集合である場合にはＳ３００の処理に進み、これ以外の場合にはＳ２８８の処理に進む。

【０１０２】

ステップ２８８（Ｓ２８８）において、プログラム実行装置１は、ベーシックブロックを１つ生成する。

【０１０３】

ステップ２９０（Ｓ２９０）において、プログラム実行装置１は、新たに生成したベーシックブロックからループ開始点への制御フロー上の経路を生成する。

【０１０４】

ステップ２９２（Ｓ２９２）において、プログラム実行装置１は、ループ外からループ開始点への経路を、新たに作成したベーシックブロックへの経路に変更する。

【０１０５】

ステップ２９４（Ｓ２９４）において、プログラム実行装置１は、Ｓ２９８までのループ３の処理を、再計算を行う各ローカル変数に対して繰り返す。

【０１０６】

ステップ２９６（Ｓ２９６）において、プログラム実行装置１は、再計算情報を用いて再計算コードを生成し、新たに生成したベーシックブロックのコードに追加する。

【０１０７】

ステップ３０２（Ｓ３０２）の処理において、プログラム実行装置１は、ループ変形による遷移点のループ外への移動処理（Ｓ３２；図７）に進む。

【０１０８】

図７は、図４に示したループ変形による遷移点のループ外への移動処理（Ｓ３２）を示すフローチャートである。

【０１０９】

図７に示すように、Ｓ３２のステップ３２２（Ｓ３２２）において、プログラム実行装置１は、Ｓ２３４までのループ１の処理を、各遷移点ごとに繰り返す。

【０１１０】

ステップ３２４（Ｓ３２４）において、プログラム実行装置１は、遷移点が最適化可能なループの中に位置するか否かを判断し、位置する場合にはＳ２８６の処理に進み、これ以外の場合にはＳ３３４の処理に進む。

【０１１１】

ステップ３２６（Ｓ３２６）において、プログラム実行装置１は、ループ開始点と遷移点をともにイミディエイトリィポストドミネート(immediatly post-dom

inate)するループ内のベーシックブロックＢＢｐを求める。

〔０１１２〕

ステップ３２８（Ｓ３２８）において、プログラム実行装置１は、ループ開始点からＢＢｐに至るすべての経路の和で構成される制御フローグラフの複製を生成する。

〔０１１３〕

ステップ３３０（Ｓ３３０）において、プログラム実行装置１は、ループ外からループ開始点に接続している全ての制御フローを、複製したフローグラフでループ開始点に対応するベーシックブロックへの制御フローに変更する。

〔０１１４〕

ステップ３３２（Ｓ３３２）において、プログラム実行装置１は、複製した制御フローグラフからＢＢｐへの制御フローはそのまま残す。

〔０１１５〕

ステップ３３６（Ｓ３３６）において、プログラム実行装置１は、図４に示したＳ２０のＳ２０８の処理に戻る。

〔０１１６〕

再び図４を参照する。

ステップ２０８（Ｓ２０８）の処理において、プログラム実行装置１は、遷移点を考慮したリダンダンシイエリミネーション(redundancy elimination)を行う。

〔０１１７〕

ステップ２１０（Ｓ２１０）の処理において、プログラム実行装置１は、遷移点を考慮する必要のない最適化処理を行う。

〔０１１８〕

ステップ２１２（Ｓ２１２）において、プログラム実行装置１は、メソッドに対するネィティブコードの生成を行い、遷移情報の生成処理（Ｓ３４；図８）に進む。

〔０１１９〕

図８は、図４に示した遷移情報の生成処理（Ｓ３４）を示すフローチャートで

ある。

Ｓ３４のステップ３４２（Ｓ３４２）において、プログラム実行装置１は、コンパイルされたコードで使用するスタックフレームのサイズをメソッドからアクセスできる場所に格納する。

〔０１２０〕

ステップ３４４（Ｓ３４４）において、プログラム実行装置１は、Ｓ３２８までのループ１の処理を、各遷移点ごとに繰り返す。

〔０１２１〕

ステップ３４６（Ｓ３４６）において、プログラム実行装置１は、遷移情報（遷移点のコードアドレス、リダンダンシィエリミネーションによるキャッシュの再計算情報、遷移点のレジスタ使用状態情報、インタプリタが遷移を決定した命令のアドレス）を生成する。

〔０１２２〕

ステップ３５０（Ｓ３５０）において、プログラム実行装置１は、図３に示したＳ１４４（図３）の処理に戻る。

〔０１２３〕

［プログラム実行装置１の効果］

プログラム実行装置１が最適化する「途中から実行を遷移することが性能に大きく貢献するメソッド」は、たとえばループを含むメソッドで、そのループが無限ループであったり、多数回まわり、インタプリタ実行とコンパイルコードの実行で優位に差が出てしまったりするメソッドがあげられる。

実行せずにループを含むかどうかを調べるには実行前にコードを走査する必要がある。しかしながらコードサイズが大きい場合などは走査にかかる時間自体が性能低下の要因となりうる。

〔０１２４〕

プログラム実行装置１では、このような走査をせずに実行しながらループを検出し、コンパイルコードに遷移することで高速な実行を可能とする。

インタプリタからコンパイルコードへの遷移は、一般には任意の点であり、また遷移すべき状況を考慮すると遷移点がループ内に位置する場合がほとんどであ

る。しかしながら、遷移点がループ内に存在したままではそのループを最適化することが困難となることから、遷移点をループ外へ移動する処理は性能向上のために重要な処理となる。

【０１２５】

【発明の効果】

以上説明したように、本発明に係るプログラム実行方法は、切り替えを行うプログラム上に遷移点がある場合であっても、遷移点を超えたより高度な最適化が可能である。

また、本発明に係るプログラム実行方法によれば、マルチスレッド環境において複数の異なる遷移点から遷移する場合であっても、複数回のコンパイルが不要で、しかも、発生したコードに重複を生じさせることがない。

【図面の簡単な説明】

【図１】

本発明に係るプログラム実行装置の構成を示す図である。

【図２】

本発明に係るプログラム実行装置の処理を示す第１のフローチャートである。

【図３】

本発明に係るプログラム実行装置の処理を示す第２のフローチャートである。

【図４】

本発明に係るプログラム実行装置の処理を示す第３のフローチャートである。

【図５】

本発明に係るプログラム実行装置の処理を示す第４のフローチャートである。

【図６】

本発明に係るプログラム実行装置の処理を示す第５のフローチャートである。

【図７】

本発明に係るプログラム実行装置の処理を示す第６のフローチャートである。

【図８】

本発明に係るプログラム実行装置の処理を示す第７のフローチャートである。

【符号の説明】

１・・・コンパイラ

１０・・・サーバ

１００・・・Ｊａｖａソースコード

１０２・・・Ｊａｖａバイトコードコンパイラ（ＪＡＶＡＣ）

１０４・・・Ｊａｖａバイトコード

１２・・・クライアント

１４０・・・記録媒体

１４・・・記憶装置

１３２・・・ハードウェア

１２０・・・Ｊａｖａ処理プログラム

１２２・・・Ｊａｖａバイトコードベリファイヤ

１２４・・・Ｊａｖａインタプリタ

１２６・・・ＪＩＴコンパイラ

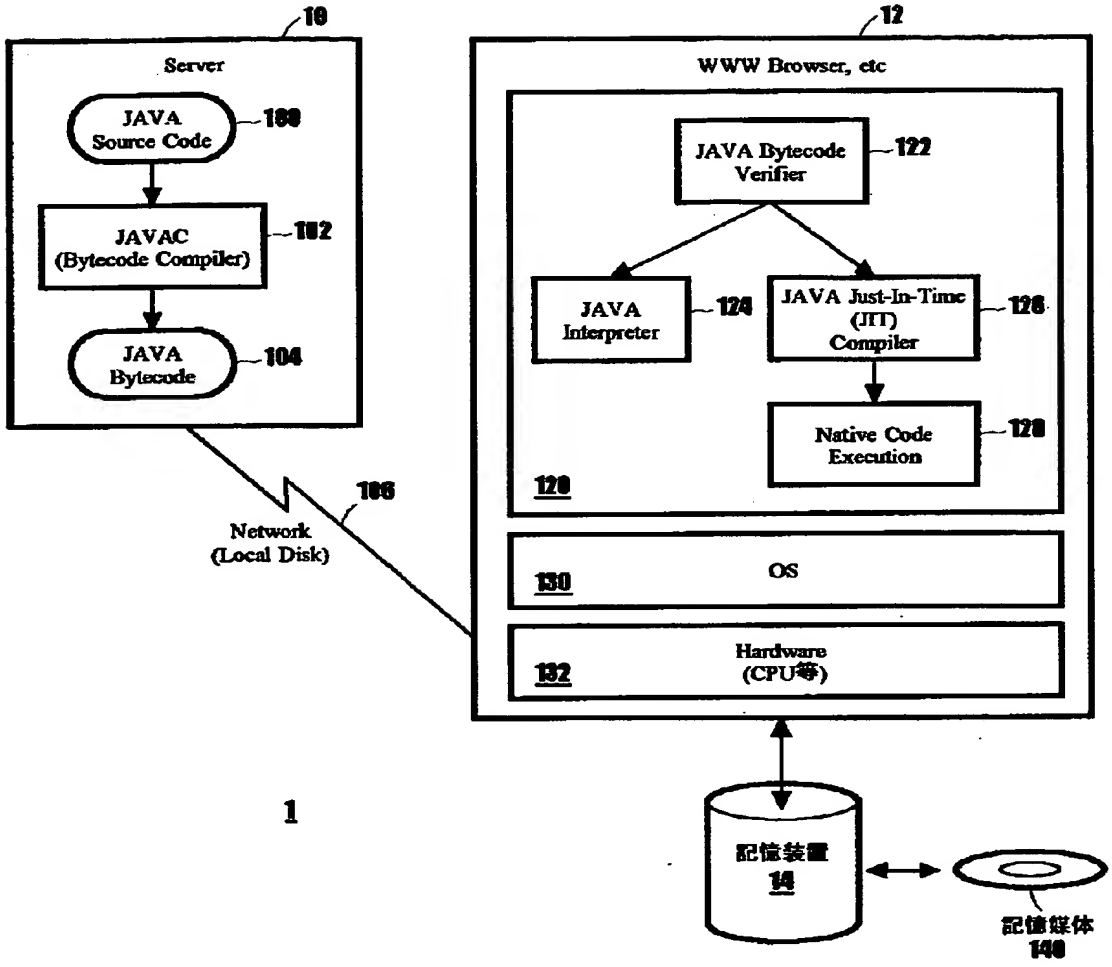
１２８・・・ネイティブコードエキスキューション

１３０・・・ＯＳ１３０

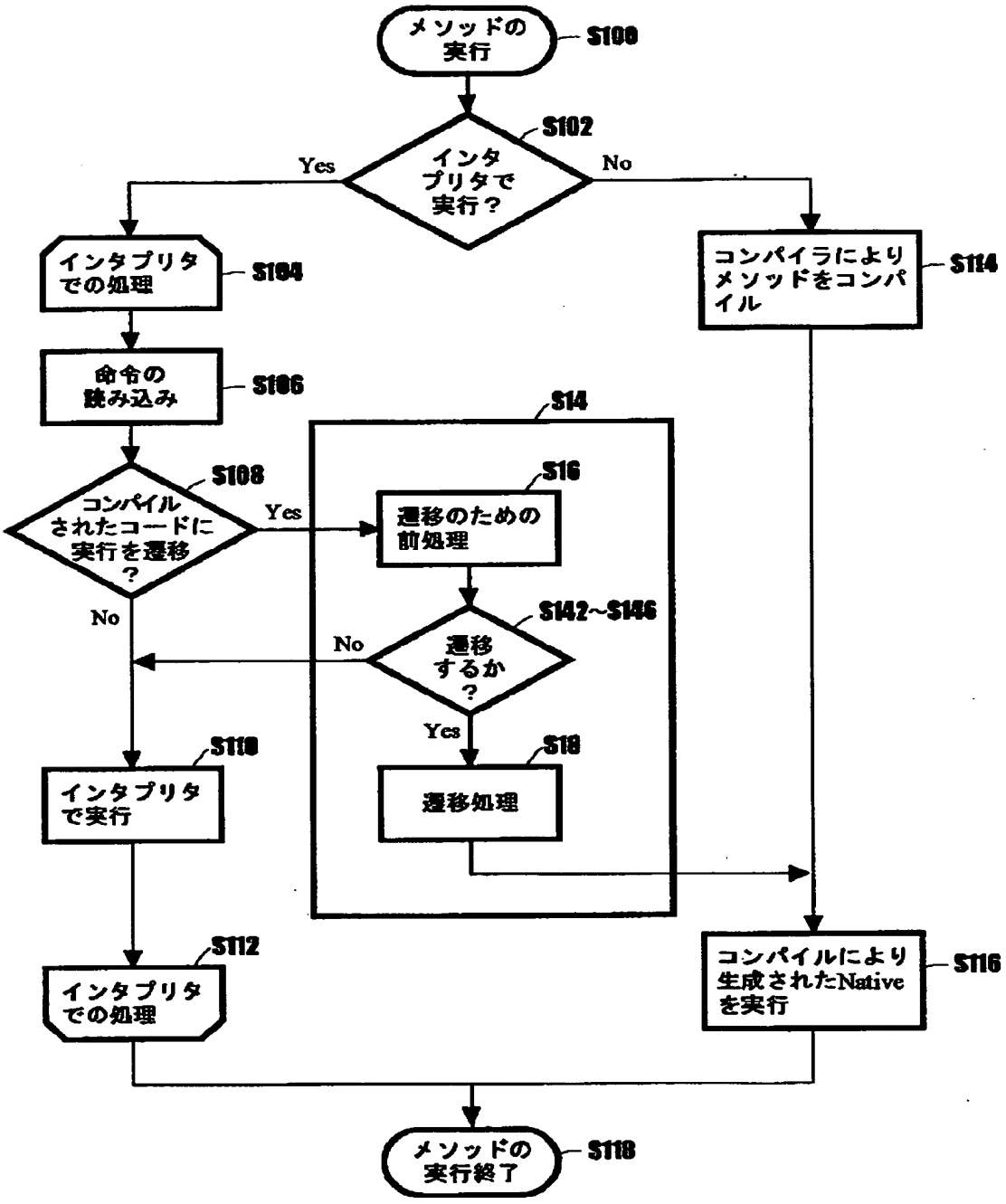
１０６・・・ネットワーク

【書類名】　　図面

【図１】

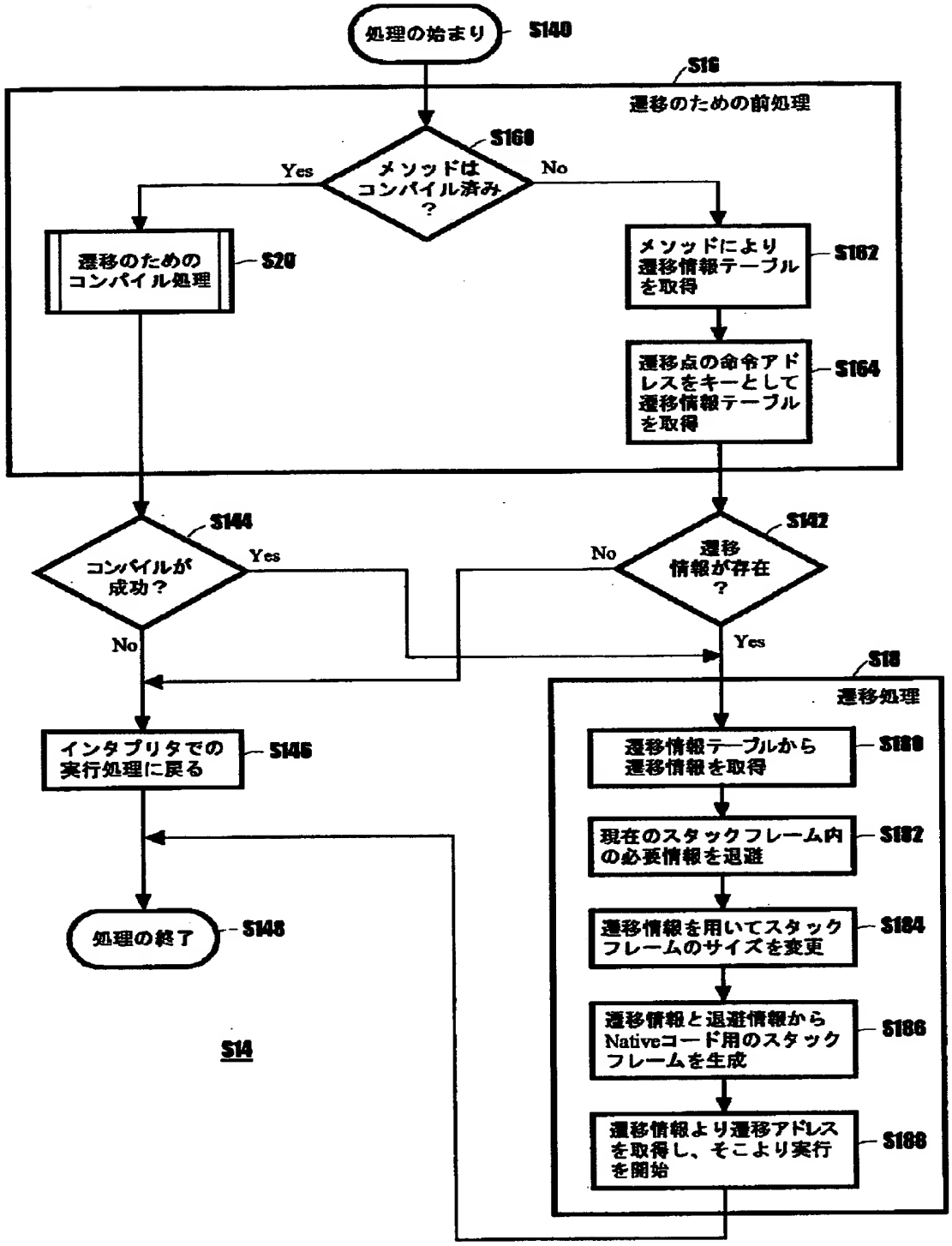

10
Server
JAVA Source Code
100
JAVAC (Bytecode Compiler)
102
JAVA Bytecode
104
106
Network (Local Disk)
12
WWW Browser, etc
JAVA Bytecode Verifier
122
JAVA Interpreter
124
JAVA Just-In-Time (JIT) Compiler
126
Native Code Execution
128
120
OS
130
Hardware (CPU等)
132
1
記憶装置
14
記憶媒体
140

【図２】

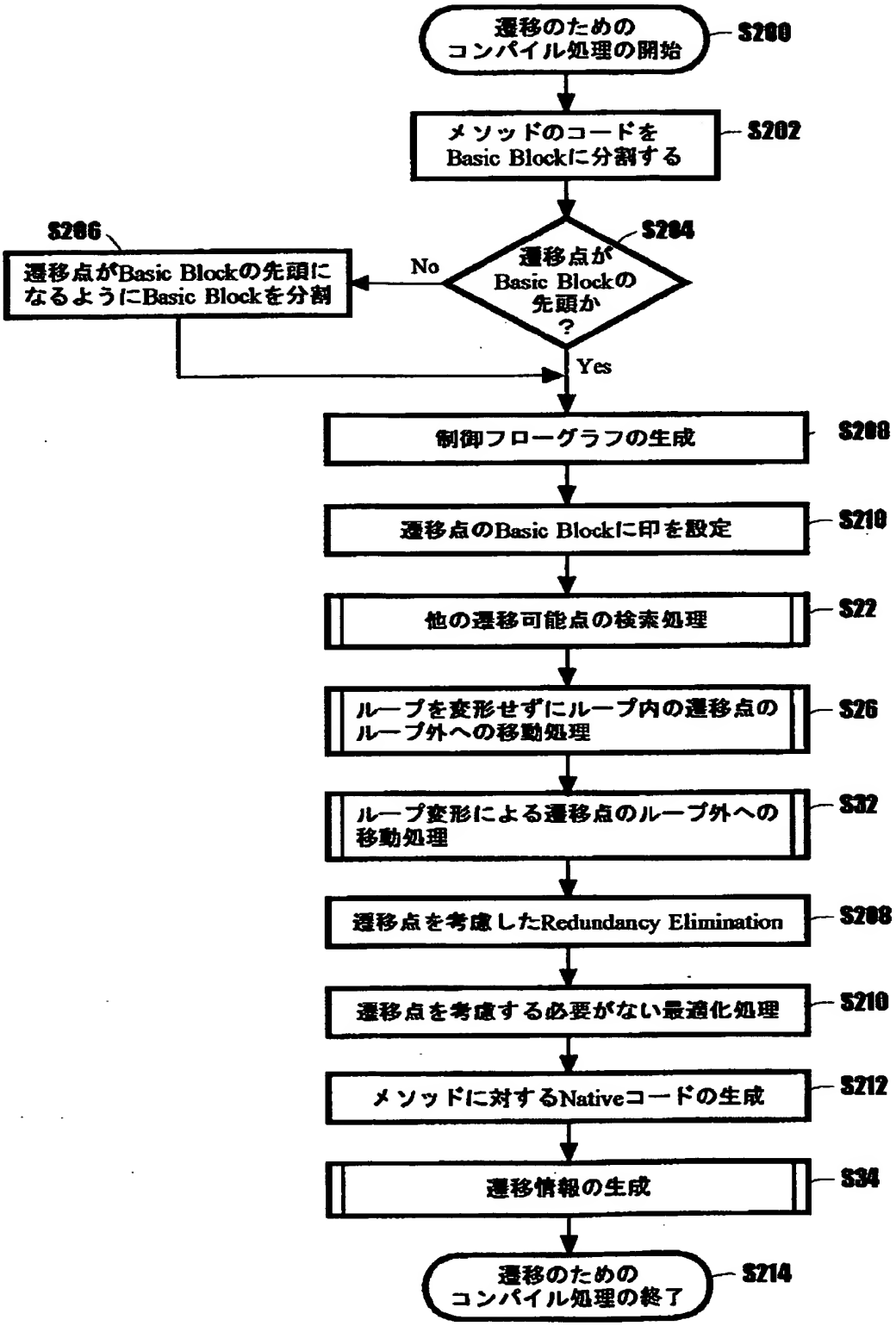

メソッドの実行
S100
S102
インタプリタで実行？
Yes
No
インタプリタでの処理
S104
コンパイラによりメソッドをコンパイル
S114
命令の読み込み
S106
S14
S108
コンパイルされたコードに実行を遷移？
Yes
S16
遷移のための前処理
No
S142~S146
遷移するか？
No
Yes
S110
インタプリタで実行
S18
遷移処理
S112
インタプリタでの処理
S116
コンパイルにより生成されたNativeを実行
メソッドの実行終了
S118

S10

【図３】

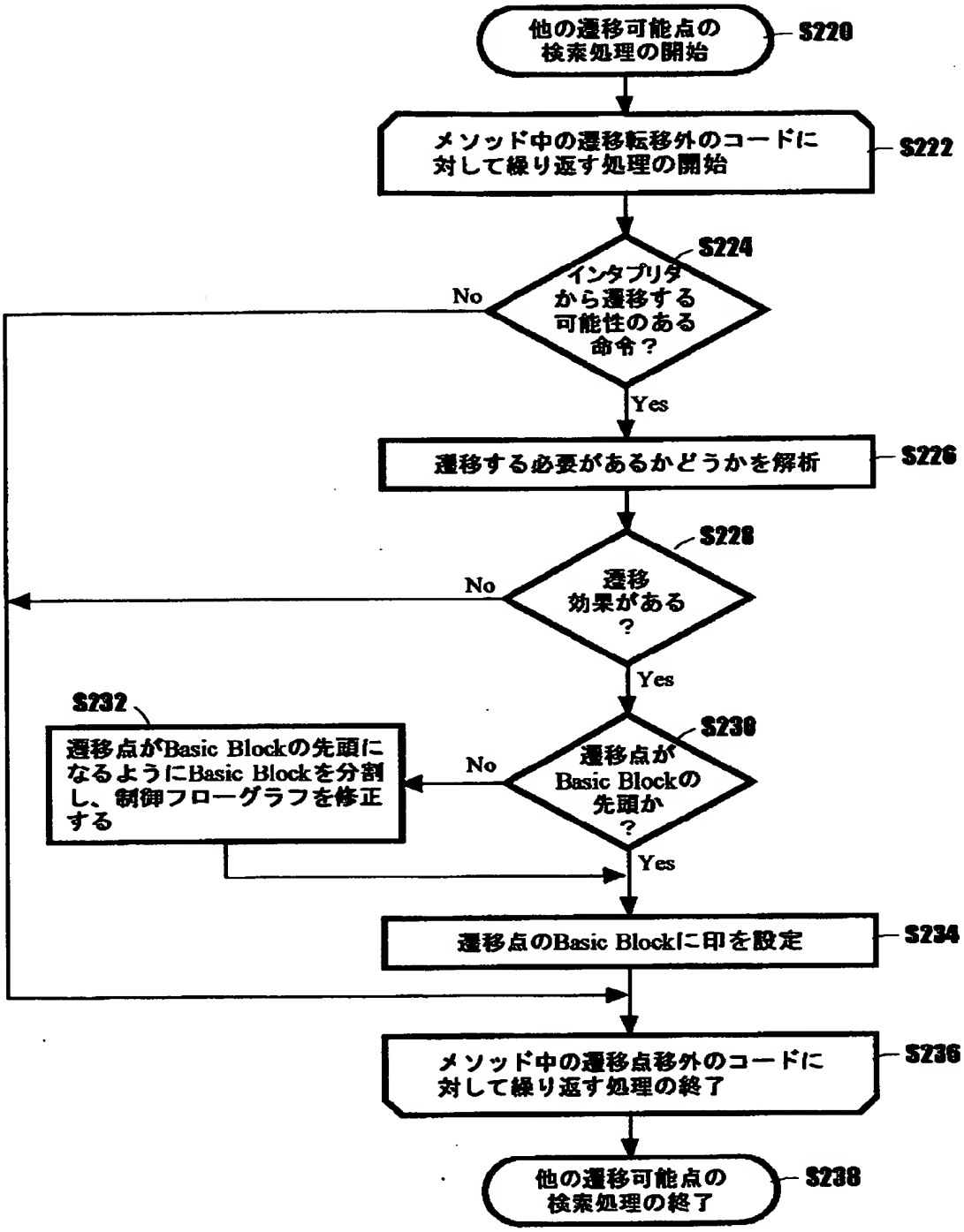

処理の始まり
S140
S16
遷移のための前処理
S160
メソッドはコンパイル済み？
Yes
No
遷移のためのコンパイル処理
S20
メソッドにより遷移情報テーブルを取得
S162
遷移点の命令アドレスをキーとして遷移情報テーブルを取得
S164
S144
コンパイルが成功？
Yes
No
S142
遷移情報が存在？
No
Yes
S18
遷移処理
インタプリタでの実行処理に戻る
S146
遷移情報テーブルから遷移情報を取得
S180
現在のスタックフレーム内の必要情報を退避
S182
遷移情報を用いてスタックフレームのサイズを変更
S184
遷移情報と退避情報からNativeコード用のスタックフレームを生成
S186
遷移情報より遷移アドレスを取得し、そこより実行を開始
S188
処理の終了
S148
S14

【図４】


遷移のための
コンパイル処理の開始
S200
メソッドのコードを
Basic Blockに分割する
S202
S204
遷移点が
Basic Blockの
先頭か
？
No
S206
遷移点がBasic Blockの先頭に
なるようにBasic Blockを分割
Yes
制御フローグラフの生成
S208
遷移点のBasic Blockに印を設定
S210
他の遷移可能点の検索処理
S22
ループを変形せずにループ内の遷移点の
ループ外への移動処理
S26
ループ変形による遷移点のループ外への
移動処理
S32
遷移点を考慮したRedundancy Elimination
S208
遷移点を考慮する必要がない最適化処理
S210
メソッドに対するNativeコードの生成
S212
遷移情報の生成
S34
遷移のための
コンパイル処理の終了
S214

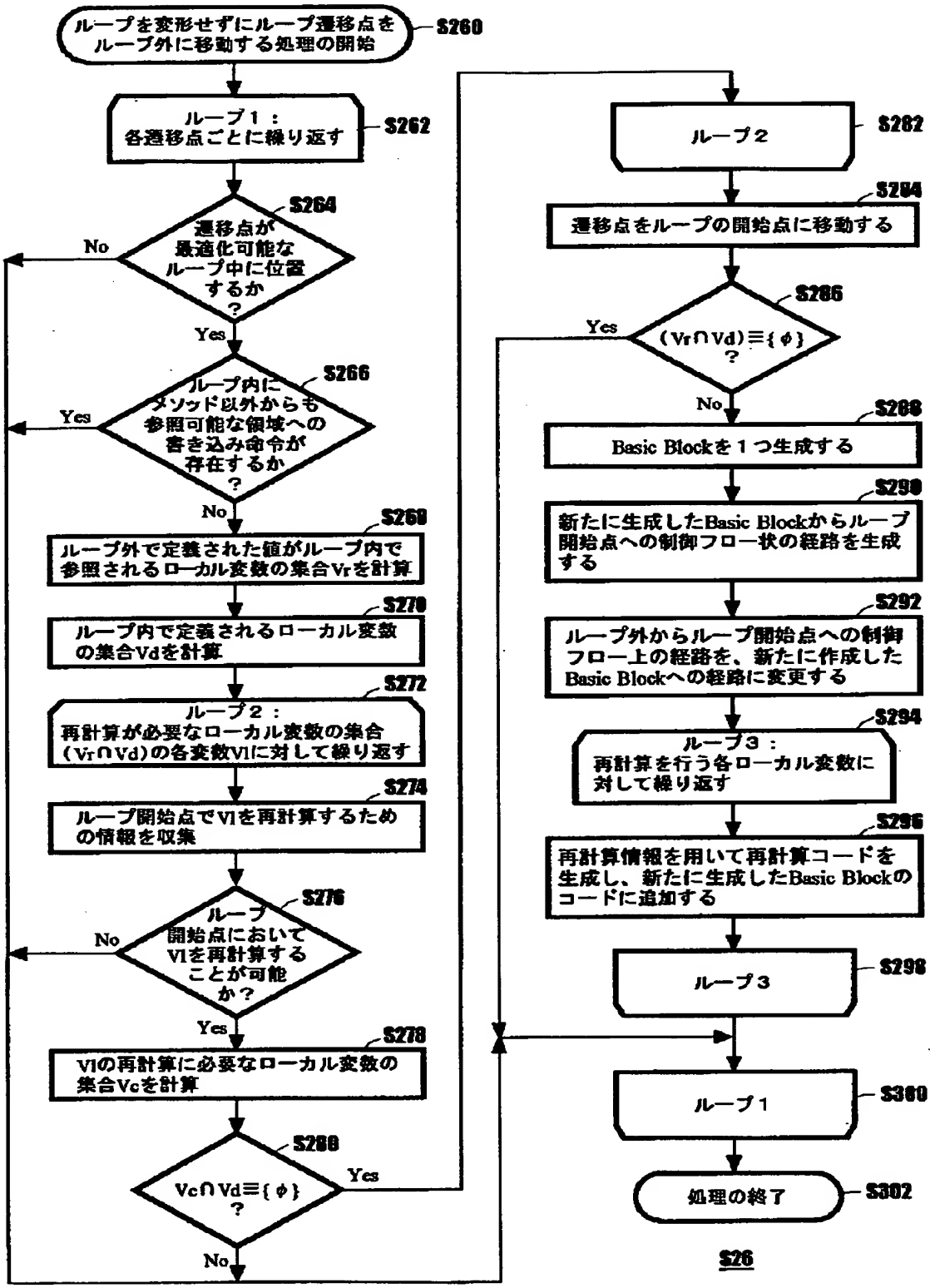
S20

【図５】

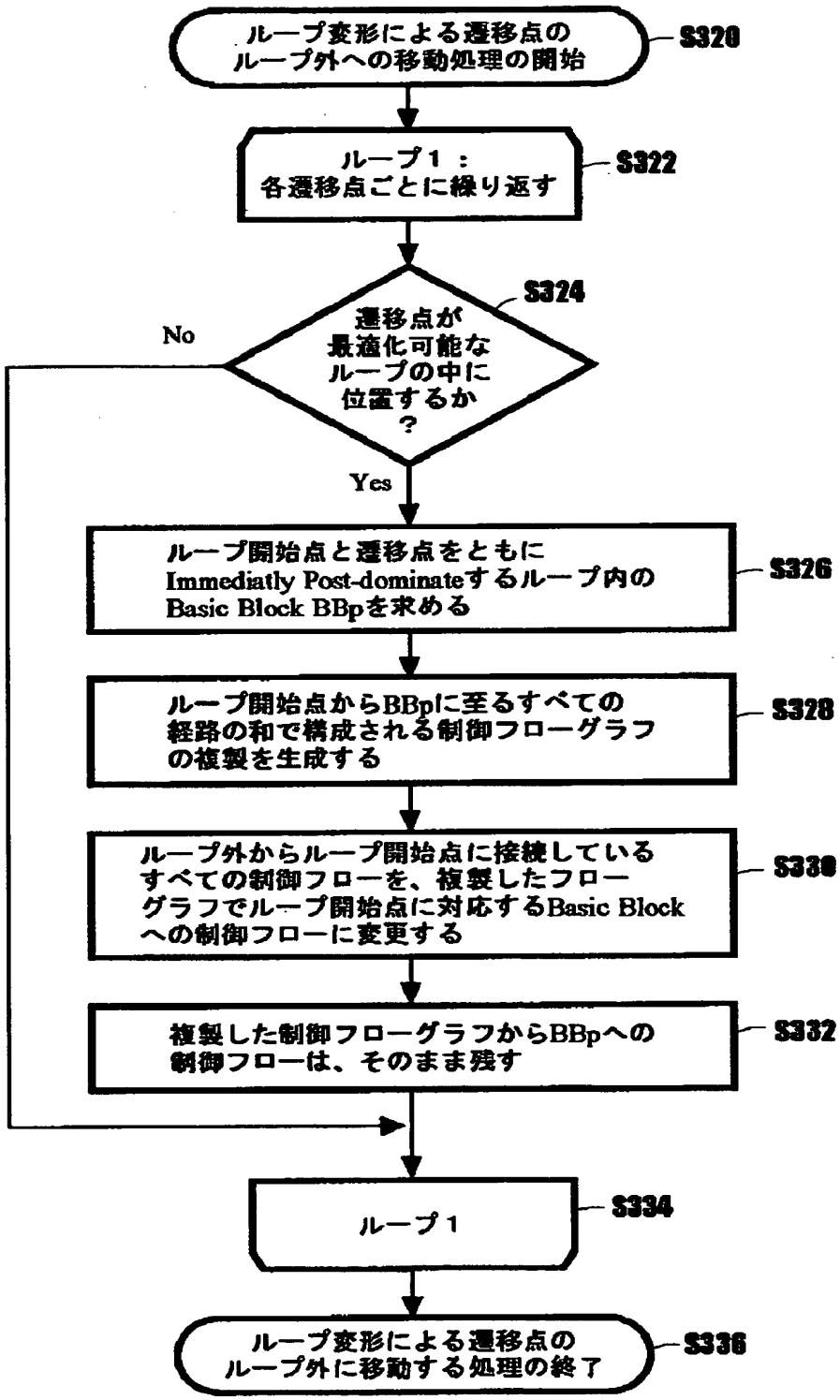

他の遷移可能点の検索処理の開始
S220
メソッド中の遷移転移外のコードに対して繰り返す処理の開始
S222
S224
インタプリタから遷移する可能性のある命令？
No
Yes
遷移する必要があるかどうかを解析
S226
S228
遷移効果がある？
No
Yes
S232
遷移点がBasic Blockの先頭になるようにBasic Blockを分割し、制御フローグラフを修正する
S230
遷移点がBasic Blockの先頭か？
No
Yes
遷移点のBasic Blockに印を設定
S234
メソッド中の遷移点移外のコードに対して繰り返す処理の終了
S236
他の遷移可能点の検索処理の終了
S238

S22

【図６】

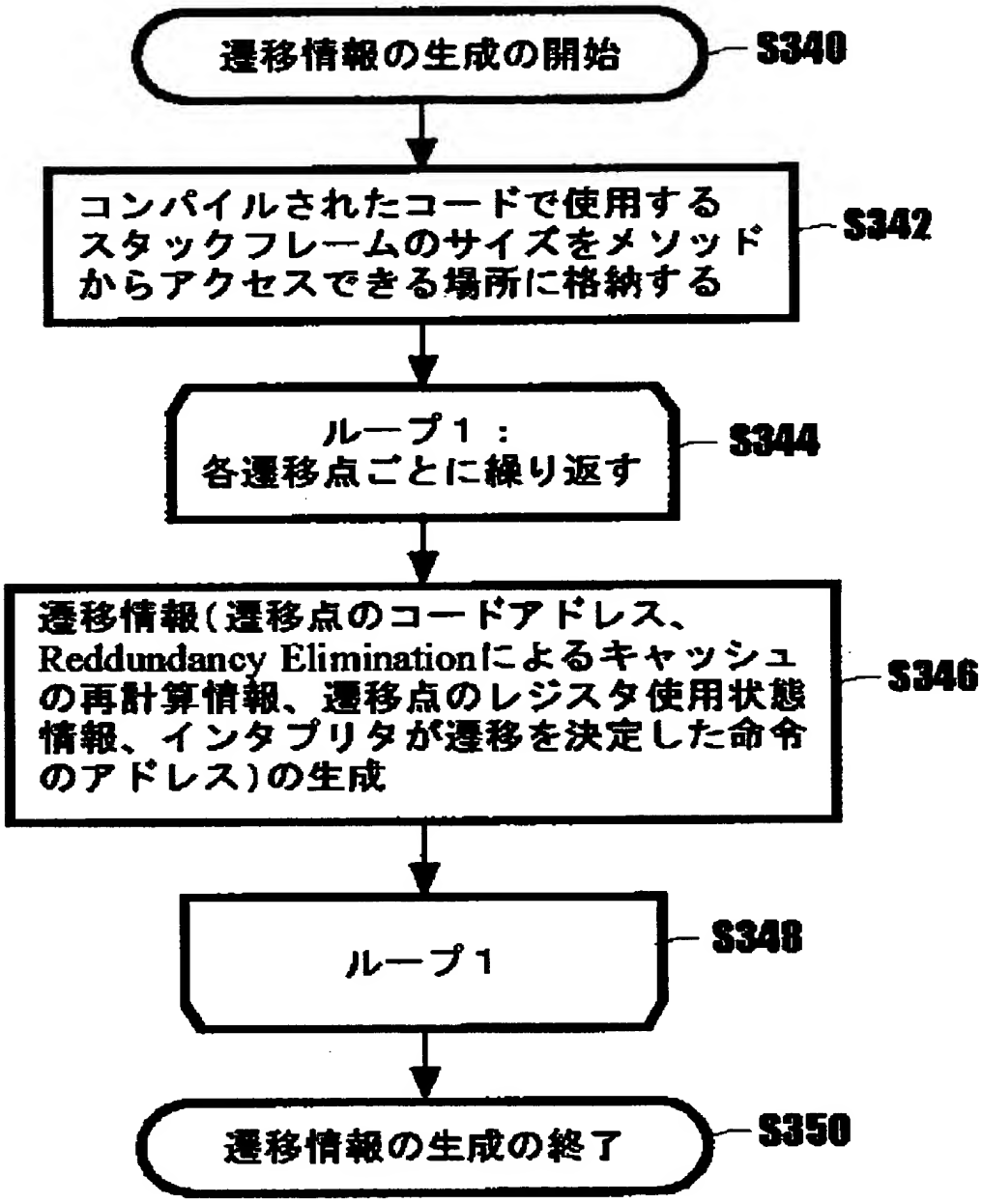

ループを変形せずにループ遷移点をループ外に移動する処理の開始
S260
ループ１：各遷移点ごとに繰り返す
S262
遷移点が最適化可能なループ中に位置するか？
S264
No
Yes
ループ内にメソッド以外からも参照可能な領域への書き込み命令が存在するか？
S266
Yes
No
ループ外で定義された値がループ内で参照されるローカル変数の集合Vrを計算
S268
ループ内で定義されるローカル変数の集合Vdを計算
S270
ループ２：再計算が必要なローカル変数の集合(Vr∩Vd)の各変数Vlに対して繰り返す
S272
ループ開始点でVlを再計算するための情報を収集
S274
ループ開始点においてVlを再計算することが可能か？
S276
No
Yes
Vlの再計算に必要なローカル変数の集合Vcを計算
S278
Vc∩Vd≡{φ}？
S280
Yes
No
ループ２
S282
遷移点をループの開始点に移動する
S284
(Vr∩Vd)≡{φ}？
S286
Yes
No
Basic Blockを１つ生成する
S288
新たに生成したBasic Blockからループ開始点への制御フロー状の経路を生成する
S290
ループ外からループ開始点への制御フロー上の経路を、新たに作成したBasic Blockへの経路に変更する
S292
ループ３：再計算を行う各ローカル変数に対して繰り返す
S294
再計算情報を用いて再計算コードを生成し、新たに生成したBasic Blockのコードに追加する
S296
ループ３
S298
ループ１
S300
処理の終了
S302
S26

【図７】


ループ変形による遷移点のループ外への移動処理の開始
S320
ループ１：各遷移点ごとに繰り返す
S322
S324
遷移点が最適化可能なループの中に位置するか？
No
Yes
ループ開始点と遷移点をともにImmediatly Post-dominateするループ内のBasic Block BBpを求める
S326
ループ開始点からBBpに至るすべての経路の和で構成される制御フローグラフの複製を生成する
S328
ループ外からループ開始点に接続しているすべての制御フローを、複製したフローグラフでループ開始点に対応するBasic Blockへの制御フローに変更する
S330
複製した制御フローグラフからBBpへの制御フローは、そのまま残す
S332
ループ１
S334
ループ変形による遷移点のループ外に移動する処理の終了
S336

S32

【図８】


遷移情報の生成の開始
S340
コンパイルされたコードで使用するスタックフレームのサイズをメソッドからアクセスできる場所に格納する
S342
ループ１：
各遷移点ごとに繰り返す
S344
遷移情報（遷移点のコードアドレス、Reddundancy Eliminationによるキャッシュの再計算情報、遷移点のレジスタ使用状態情報、インタプリタが遷移を決定した命令のアドレス）の生成
S346
ループ１
S348
遷移情報の生成の終了
S350

S34

【書類名】　要約書

【要約】

【課題】より高度な最適化が可能なプログラム実行方法を提供する。

【解決手段】　本発明に係るプログラム実行装置１は、メソッドを実行する途中で、コンパイル処理とインタプリタ処理との間で遷移を行う。この際、ループ入り口に遷移点を移動させても実行上の問題がない場合には、コードの遷移点をループの入り口に移動させ、遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製し、再計算コードを生成するための情報を遷移点に持たせ、再計算を実行する。

【選択図】　図１

認定・付加情報

特許出願の番号	平成１１年　特許願　第３２６９９０号
受付番号	５９９０１１２４７９３
書類名	特許願
担当官	第七担当上席　　００９６
作成日	平成１１年１１月１９日

＜認定情報・付加情報＞

【提出日】　平成11年11月17日

次頁無

出証特２０００－３０２１１５１

出　願　人　履　歴　情　報

識別番号　　〔３９０００９５３１〕

１．変更年月日　　１９９０年１０月２４日
［変更理由］　新規登録
住　所　　アメリカ合衆国１０５０４、ニューヨーク州　アーモンク　（番地なし）
氏　名　　インターナショナル・ビジネス・マシーンズ・コーポレイション